Exhibit 4

U.S. Patent No. 7,784,058 vs. Oracle

Accused Instrumentalities: Oracle products and services using user mode critical system elements as shared libraries, including without limitation Oracle Cloud Infrastructure ("OCI") and Oracle Kubernetes Engine ("OCE"), and all versions and variations thereof since the issuance of the asserted patent.

Claim 1

Claim 1	Accused Instrumentalities
[1pre] 1. A computing system for executing a plurality of software applications comprising:	To the extent the preamble is limiting, each Accused Instrumentality comprises or constitutes a computing system for executing a plurality of software applications as claimed. *See* claim limitations below. *See also, e.g.*:

Claim 1	Accused Instrumentalities
	Why Choose OKE? **Price-Performance** OKE is the lowest cost Kubernetes service amongst all hyperscalers, especially for serverless. **Autoscaling** OKE automatically adjusts compute resources based on demand, which can reduce your costs. **Efficiency** GPUs can be scarce, but OKE job scheduling makes it easy to maximize resource utilization. **Portability** OKE is consistent across clouds and on-premises, enabling portability and avoiding vendor lock-in. **Simplicity** OKE reduces the time and cost needed to manage the complexities of Kubernetes infrastructure. **Reliability** Automatic upgrades and security patching boost reliability for the control plane and worker nodes. https://www.oracle.com/cloud/cloud-native/kubernetes-engine/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
<h1>Welcome to Oracle Cloud Infrastructure</h1>
Oracle Cloud Infrastructure (OCI) is a set of complementary cloud services that enable you to build and run a range of applications and services in a highly available hosted environment. OCI provides high-performance compute capabilities (as physical hardware instances) and storage capacity in a flexible overlay virtual network that is securely accessible from your on-premises network.
https://docs.oracle.com/en-us/iaas/Content/GSG/Concepts/baremetalintro.htm
<h2>Existing applications can benefit by migrating to OCI and OKE</h2>
OKE offers lower total cost of ownership and improved time to market.
OKE simplifies operations at scale in the following ways:

Lift and shift; there's no need to rearchitect
Reduce operations burden with automation
Save time on infrastructure management
Increase resource utilization and efficiency
Improve agility, flexibility, uptime, and resilience
Reduce compliance risk and enhance security

https://www.oracle.com/cloud/cloud-native/kubernetes-engine/#app-migration
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	What is OCI Kubernetes Engine (OKE)? Oracle Cloud Infrastructure Kubernetes Engine (OKE) is a managed Kubernetes service that simplifies the development, deployment, and operation of containerized workloads at scale. OKE enables you to quickly create, manage, and consume Kubernetes clusters that leverage underlying OCI compute, networking, and storage services. **When should I use OKE?** You should use OKE when you want to leverage Kubernetes to deploy and manage your Kubernetes-based container applications. It allows you to combine the production-grade container orchestration of standard upstream Kubernetes with the control, security, and high, predictable performance of OCI. https://www.oracle.com/cloud/cloud-native/kubernetes-engine/faq/ **How does OKE provide resiliency?** When you create an OKE cluster, OKE automatically creates and manages multiple Kubernetes control plane nodes spread across fault domains and availability domains (logical data centers). This is done to help ensure that the managed Kubernetes control plane is highly available. Control plane operations, such as upgrading to newer versions of Kubernetes, can be performed without service interruptions. Additionally, when you provision worker nodes, you can use a placement configuration to control the fault domain and availability domain where they are created. Nodes will automatically come online with labels, which you can use to schedule your workloads so they are robust and highly available. https://www.oracle.com/cloud/cloud-native/kubernetes-engine/faq/ **Can I deploy private Kubernetes clusters?** Yes; with OKE, your Kubernetes clusters are integrated in your VCN. Your cluster worker nodes, load balancers, and the Kubernetes API endpoint are part of a private or public subnet of your VCN. Regular VCN routing and firewall rules control access to the Kubernetes API endpoint, making it accessible from a corporate network only, through a bastion host, or by specific platform services. https://www.oracle.com/cloud/cloud-native/kubernetes-engine/faq/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
When should I use virtual nodes, managed nodes, or self-managed nodes?

Virtual nodes
Virtual nodes offer a serverless Kubernetes experience. This option is ideal if you'd rather focus on your application and avoid managing the underlying infrastructure. Virtual nodes relieve you of management-related tasks such as scaling, upgrading, patching, troubleshooting, and provisioning worker nodes.
Managed nodes
Managed nodes are a good choice for general purpose workloads. They offer an extensive list of customizable configuration options that have been tested by the OKE service. Unlike fully managed virtual nodes, you share the management of worker nodes with OCI. OKE simplifies the management process through features such as on-demand cycling to automate worker node updates, cluster self-healing upon failure detection, autoscaling, and more.
Self-managed nodes
Self-managed nodes offer access to the underlying infrastructure, configuration options, and compute shapes that aren't currently available to managed nodes. This includes access to specialized infrastructure, such as RDMA-enabled bare metal cluster networks or confidential compute shapes. This advanced control makes self-managed nodes ideal for specialized use cases that aren't supported with managed nodes. Note that with self-managed nodes, you are fully responsible for managing the worker nodes—without the automated features provided by managed or virtual nodes.

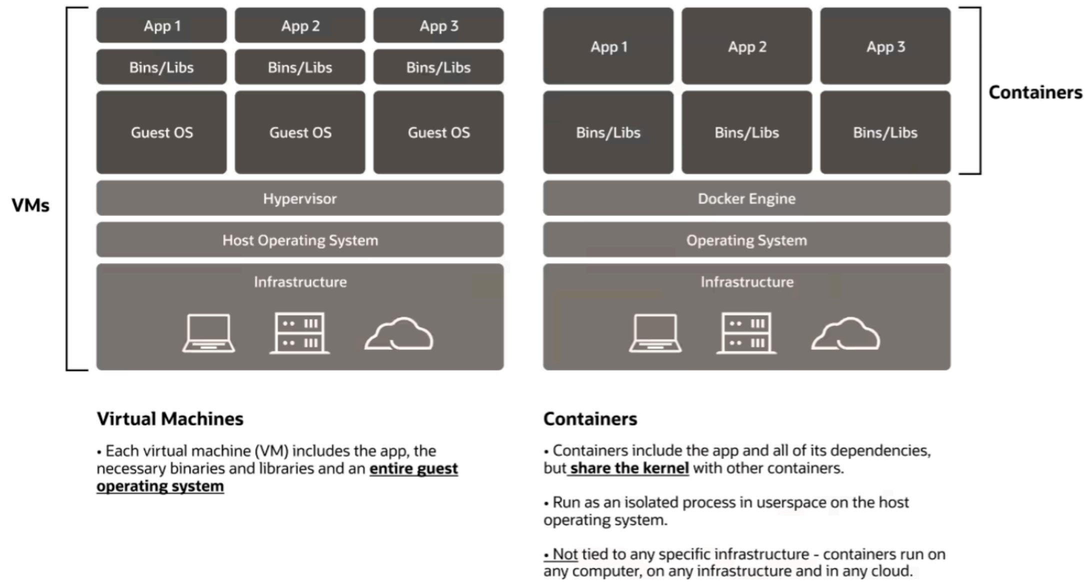
https://www.oracle.com/cloud/cloud-native/kubernetes-engine/faq/

What are the storage options for virtual nodes?
OKE virtual nodes do not yet have persistent storage capabilities. However, there are plans to introduce support for attaching persistent volumes backed by OCI Block Storage and OCI File Storage. If your Kubernetes application requires persistent storage, it's advisable to use OKE managed nodes.

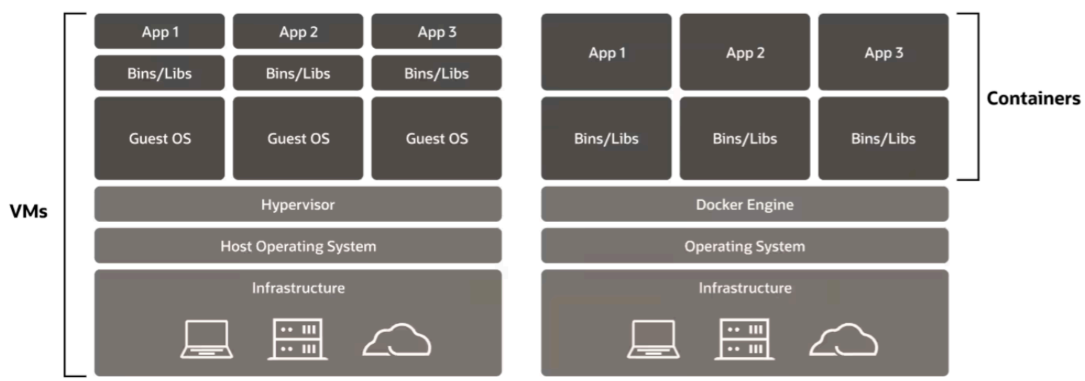
https://www.oracle.com/cloud/cloud-native/kubernetes-engine/faq/
</td></tr>
</table>

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
<h2>Supported Images for Managed Nodes</h2>
Kubernetes Engine supports the provisioning of worker nodes (managed nodes only) using some, but not all, of the latest Oracle Linux images provided by Oracle Cloud Infrastructure.
You can use these Oracle Linux images when provisioning managed nodes:
OKE Images
Platform Images
Custom Images
https://docs.oracle.com/en-us/iaas/Content/ContEng/Reference/contengimagesshapes.htm
<h2>What is Docker?</h2>
A Docker container is a packaging format that packages all the code and dependencies of an application in a standard format that allows it to run quickly and reliably across computing environments. A Docker container is a popular lightweight, standalone, executable container that includes everything needed to run an application, including libraries, system tools, code, and runtime. Docker is also a software platform that allows developers to build, test, and deploy containerized applications quickly.
Containers as a Service (CaaS) or Container Services are managed cloud services that manage the lifecycle of containers. Container services help orchestrate (start, stop, scale) the runtime of containers. Using container services, you can simplify, automate, and accelerate your application development and deployment lifecycle.
Docker and Container Services have seen rapid adoption and have been a tremendous success over the last several years. From an almost unknown and rather technical open source technology in 2013, Docker has evolved into a standardized runtime environment now officially supported for many Oracle enterprise products.
https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Container: Unlike a VM which provides hardware virtualization, a container provides lightweight, operating-system-level virtualization by abstracting the "user space." Containers share the host system's kernel with other containers. A container, which runs on the host operating system, is a standard software unit that packages code and all its dependencies, so applications can run quickly and reliably from one environment to another. Containers are nonpersistent and are spun up from images. **Docker engine:** The open source host software building and running the containers. Docker Engines act as the client-server application supporting containers on various Windows servers and Linux operating systems, including Oracle Linux, CentOS, Debian, Fedora, RHEL, SUSE, and Ubuntu. **Docker images:** Collection of software to be run as a container that contains a set of instructions for creating a container that can run on the Docker platform. Images are immutable, and changes to an image require to build a new image. **Docker Registry:** Place to store and download images. The registry is a stateless and scalable server-side application that stores and distributes Docker images. https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/ **Docker versus Kubernetes** Linux containers have existed since 2008, but they were not well known until the emergence of Docker containers in 2013. With the onset of Docker containers, came the explosion of interest in developing and deploying containerized applications. As the number of containerized applications grew to span hundreds of containers deployed across multiple servers, operating them became more complex. How do you coordinate, scale, manage, and schedule hundreds of containers? This is where Kubernetes can help. Kubernetes is an open source orchestration system that allows you to run your Docker containers and workloads. It helps you manage the operating complexities when moving to scale multiple containers deployed across multiple servers. The Kubernetes engine automatically orchestrates the container lifecycle, distributing the application containers across the hosting infrastructure. Kubernetes can quickly scale resources up or down, depending on the demand. It continually provisions, schedules, deletes, and monitors the health of the containers. https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/

Claim 1	Accused Instrumentalities
	Docker Basics The core concepts of Docker are images and containers. A Docker image contains everything that is needed to run your software: the code, a runtime (for example, Java Virtual Machine (JVM), drivers, tools, scripts, libraries, deployments, and more. A Docker container is a running instance of a Docker image. However, unlike in traditional virtualization with a type 1 or type 2 hypervisor, a Docker container runs on the kernel of the host operating system. Within a Docker image there is no separate operating system, as illustrated in Figure 1.  https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/

Claim 1	Accused Instrumentalities
	Container Cloud Services The first part of this article explained some important Docker concepts. However, in a production environment it is not enough to simply run an application in a Docker container. To setup and operate a production environment requires hardware to run the containers. Software such as Docker, along with repositories and cluster managers, must be installed, upgraded and patched. If several Docker containers communicate across hosts, a network must be created. Clustered containers should be restarted if they fail. In addition, a set of containers linked to each other should be deployable as easily as a single logical application instance. An example of this could be a load balancer, a few web servers, some Oracle WebLogic Server instances with an admin server, a managed server, and a database. To manage containerized applications at scale, requires a container orchestration system like Kubernetes or Docker Swarm. Deploying, managing, and operating orchestration systems like Kubernetes can be challenging and time-consuming. To make it easier and more efficient for developers to create containerized applications, cloud providers offer Container Cloud Services or Containers as a Service (CaaS). Container Cloud Services help developers and operations teams streamline and manage the lifecycle of containers in an automated fashion. These orchestration services, typically built using Kubernetes, make it easier for DevOps teams to manage and operate containerized applications at scale. Oracle Cloud Infrastructure Kubernetes Engine and Azure Kubernetes Service are two examples of popular container orchestration managed cloud services. Oracle Cloud Infrastructure Kubernetes Engine is a fully managed, scalable, and highly available service that you can use to deploy your containerized applications in the cloud. Use Kubernetes Engine (sometimes abbreviated to just OKE) when your development team wants to reliably build, deploy, and manage cloud native applications. https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/

Claim 1	Accused Instrumentalities
[1a] a) a processor;	Each Accused Instrumentality comprises a processor. For example, each node/host contains at least one CPU. *See, e.g.*: **Docker Basics** The core concepts of Docker are images and containers. A Docker image contains everything that is needed to run your software: the code, a runtime (for example, Java Virtual Machine (JVM), drivers, tools, scripts, libraries, deployments, and more. A Docker container is a running instance of a Docker image. However, unlike in traditional virtualization with a type 1 or type 2 hypervisor, a Docker container runs on the kernel of the host operating system. Within a Docker image there is no separate operating system, as illustrated in Figure 1.  **Virtual Machines** • Each virtual machine (VM) includes the app, the necessary binaries and libraries and an **entire guest operating system** **Containers** • Containers include the app and all of its dependencies, but **share the kernel** with other containers. • Run as an isolated process in userspace on the host operating system. • Not tied to any specific infrastructure - containers run on any computer, on any infrastructure and in any cloud. https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/

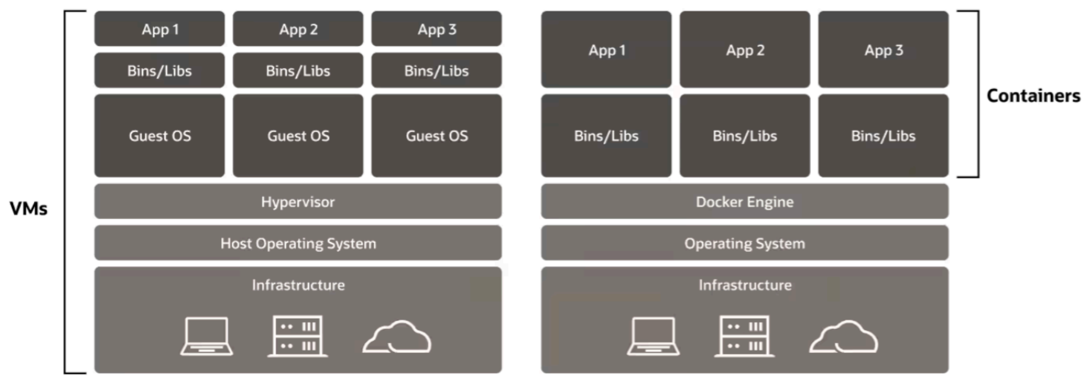
Claim 1	Accused Instrumentalities
	When should I use virtual nodes, managed nodes, or self-managed nodes? • **Virtual nodes** Virtual nodes offer a serverless Kubernetes experience. This option is ideal if you'd rather focus on your application and avoid managing the underlying infrastructure. Virtual nodes relieve you of management-related tasks such as scaling, upgrading, patching, troubleshooting, and provisioning worker nodes. • **Managed nodes** Managed nodes are a good choice for general purpose workloads. They offer an extensive list of customizable configuration options that have been tested by the OKE service. Unlike fully managed virtual nodes, you share the management of worker nodes with OCI. OKE simplifies the management process through features such as on-demand cycling to automate worker node updates, cluster self-healing upon failure detection, autoscaling, and more. • **Self-managed nodes** Self-managed nodes offer access to the underlying infrastructure, configuration options, and compute shapes that aren't currently available to managed nodes. This includes access to specialized infrastructure, such as RDMA-enabled bare metal cluster networks or confidential compute shapes. This advanced control makes self-managed nodes ideal for specialized use cases that aren't supported with managed nodes. Note that with self-managed nodes, you are fully responsible for managing the worker nodes—without the automated features provided by managed or virtual nodes. https://www.oracle.com/cloud/cloud-native/kubernetes-engine/faq/ **Welcome to Oracle Cloud Infrastructure** Oracle Cloud Infrastructure (OCI) is a set of complementary cloud services that enable you to build and run a range of applications and services in a highly available hosted environment. OCI provides high-performance compute capabilities (as physical hardware instances) and storage capacity in a flexible overlay virtual network that is securely accessible from your on-premises network. https://docs.oracle.com/en-us/iaas/Content/GSG/Concepts/baremetalintro.htm

Claim 1	Accused Instrumentalities
[1b] b) an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor; and,	Each Accused Instrumentality comprises an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor. For example, the OSCSEs include kernel-mode functions similar to the functionalities provided by user-space libraries such as glibc. These are implemented in kernel-space to handle tasks such as (without limitation) memory management (kmalloc(), kfree(), etc,) at kernel level. *See, e.g.*:

Docker Basics

The core concepts of Docker are images and containers. A Docker image contains everything that is needed to run your software: the code, a runtime (for example, Java Virtual Machine (JVM), drivers, tools, scripts, libraries, deployments, and more.

A Docker container is a running instance of a Docker image. However, unlike in traditional virtualization with a type 1 or type 2 hypervisor, a Docker container runs on the kernel of the host operating system. Within a Docker image there is no separate operating system, as illustrated in Figure 1.



Virtual Machines

• Each virtual machine (VM) includes the app, the necessary binaries and libraries and an **entire guest operating system**

Containers

• Containers include the app and all of its dependencies, but **share the kernel** with other containers.

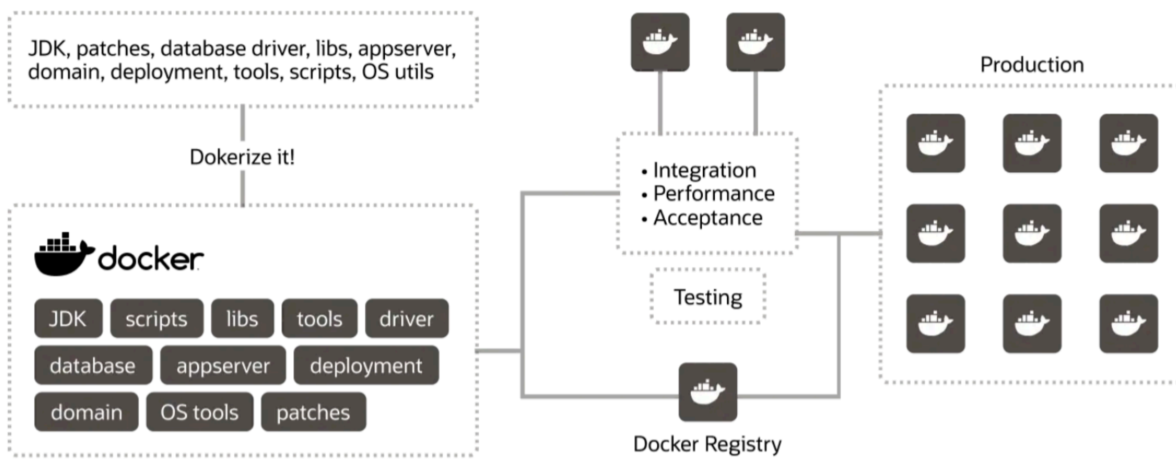
• Run as an isolated process in userspace on the host operating system.

• Not tied to any specific infrastructure - containers run on any computer, on any infrastructure and in any cloud.

https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/

Claim 1	Accused Instrumentalities
	Kernel mode Kernel mode refers to the processor mode that enables software to have full and unrestricted access to the system and its resources. The OS kernel and kernel drivers, such as the file system driver, are loaded into protected memory space and operate in this highly privileged kernel mode. https://www.techtarget.com/searchdatacenter/definition/kernel The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc

Claim 1	Accused Instrumentalities
[1c] c) a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode and	Each Accused Instrumentality comprises a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode. For example, the shared library with SLCSEs include the runtime environment, system tools, and dependencies, such as the glibc library and other libraries that replicate OSCSEs, included in the container image (including without limitation in a base image that is included within the container image). *See, e.g.*:  Figure 2 https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/

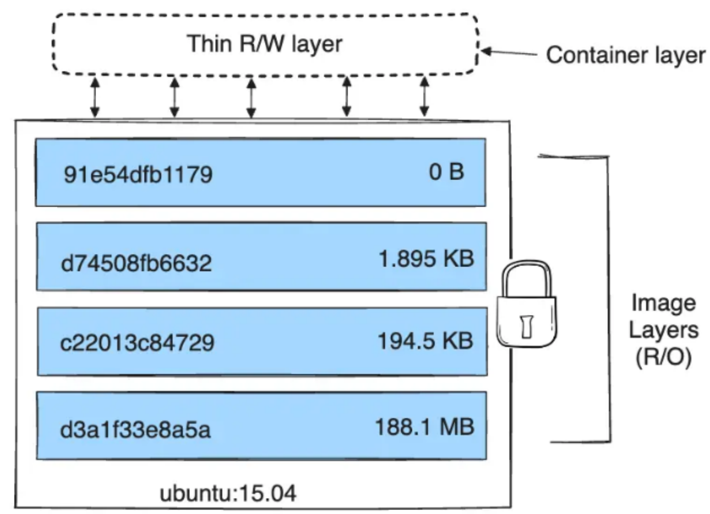
Claim 1	Accused Instrumentalities
	Docker Image—Development to Production Creating a Docker image with all of its dependencies solves the "but it worked for me on my development machine" problem. The key idea is that a Docker image is created automatically by a build pipeline from a source-code repository like Git and initially tested in a development environment. This immutable image will then be stored in a Docker registry. As shown in the Figure 4, the same image will be used for further load tests, integration tests, acceptance tests, and more. In every environment, the same image will be used. Small but necessary environmentally specific differences, such as a JDBC URL for a production database, can be fed into the container as environment variables or files.  https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/

Claim 1	Accused Instrumentalities
	Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/ Container image files are complete, static and executable versions of an application or service and differ from one technology to another. Docker images are made up of multiple layers, which start with a base image that includes all of the dependencies needed to execute code in a container. Each image has a readable/writable layer on top of static unchanging layers. Because each container has its own specific container layer that customizes that specific container, underlying image layers can be saved and reused in multiple containers. An Open Container Initiative (OCI) https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization

Claim 1	Accused Instrumentalities
	# About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities

	Images and layers
	
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	
	https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

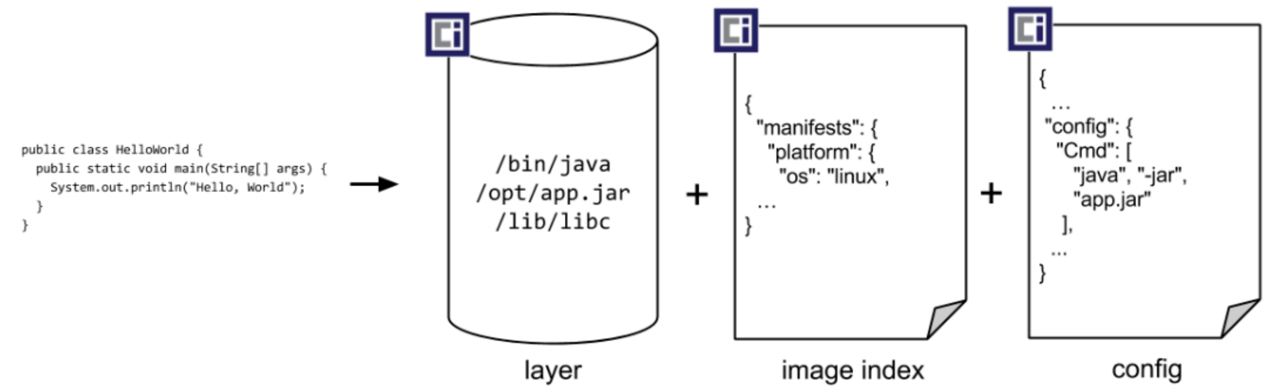
<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
<h1>Volumes</h1>
Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:
https://kubernetes.io/docs/concepts/storage/volumes/
<h2>Container environment</h2>
The Kubernetes Container environment provides several important resources to Containers:

A filesystem, which is a combination of an image and one or more volumes.
Information about the Container itself.
Information about other objects in the cluster.

https://kubernetes.io/docs/concepts/containers/container-environment/
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	Open Container Initiative **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

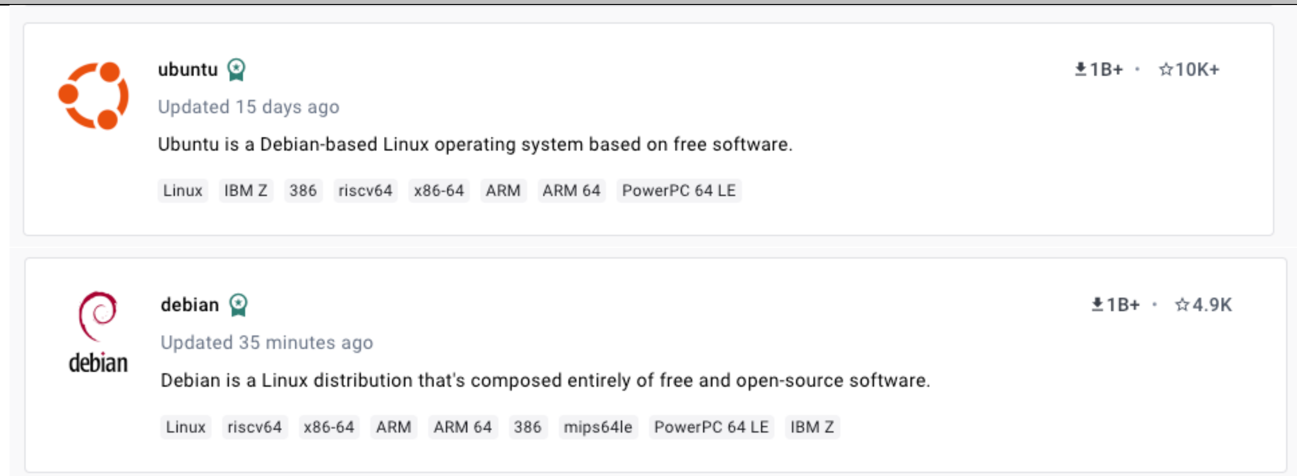
Claim 1	Accused Instrumentalities
	# OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	## Layer • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. ## Image JSON • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc

Claim 1	Accused Instrumentalities
[1d] i) wherein some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications,	In each Accused Instrumentality, some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications. For example, a base image serves as a self-contained unit that encompasses all the necessary components for an application to run, including the application code, runtime environment, system tools, and dependencies (i.e., SLCSEs). The images are based on existing Linux distributions, such as Debian and Ubuntu, including essential system elements (i.e., functional replicas of OSCSEs). Each container image is based on a specific base image, which contains the application code, and dependencies, including system libraries or shared library critical system elements (SLCSEs). The base image forms a part of the container image according to the "layer" model described in the documentation below. When the container runs the image, it creates a runtime instance of that container image. In turn, when one or more applications executes within the container runtime environment, it dynamically links to the SLCSEs stored in the runtime environment, which thereby become a part of the application(s). *See, e.g.*: **Container images** A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/

Claim 1	Accused Instrumentalities



https://hub.docker.com/search?image_filter=official&type=image&q=

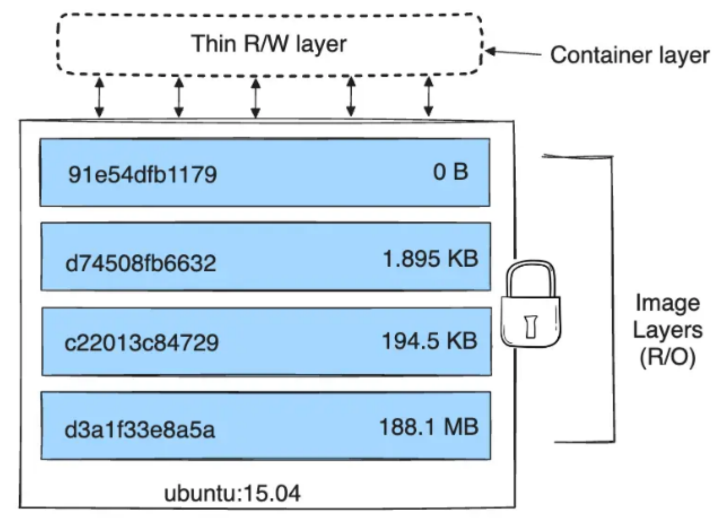
Platform	x86_64 / amd64	arm64 / aarch64	arm (32-bit)	ppc64le	s390x
CentOS	✓	✓		✓	
Debian	✓	✓	✓	✓	
Fedora	✓	✓		✓	
Raspberry Pi OS (32-bit)			✓		
RHEL (s390x)					✓
SLES					✓
Ubuntu	✓	✓	✓	✓	✓
Binaries	✓	✓	✓		

https://docs.docker.com/engine/install/

Claim 1	Accused Instrumentalities
	Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image **About storage drivers** To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities

	Images and layers
	
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	
	https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/ **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

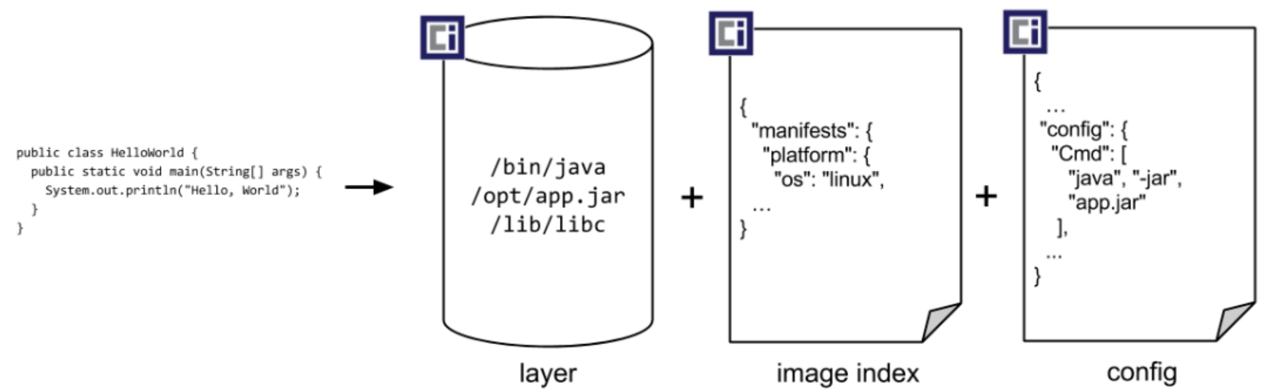
Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	Open Container Initiative **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Overview

At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.



https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

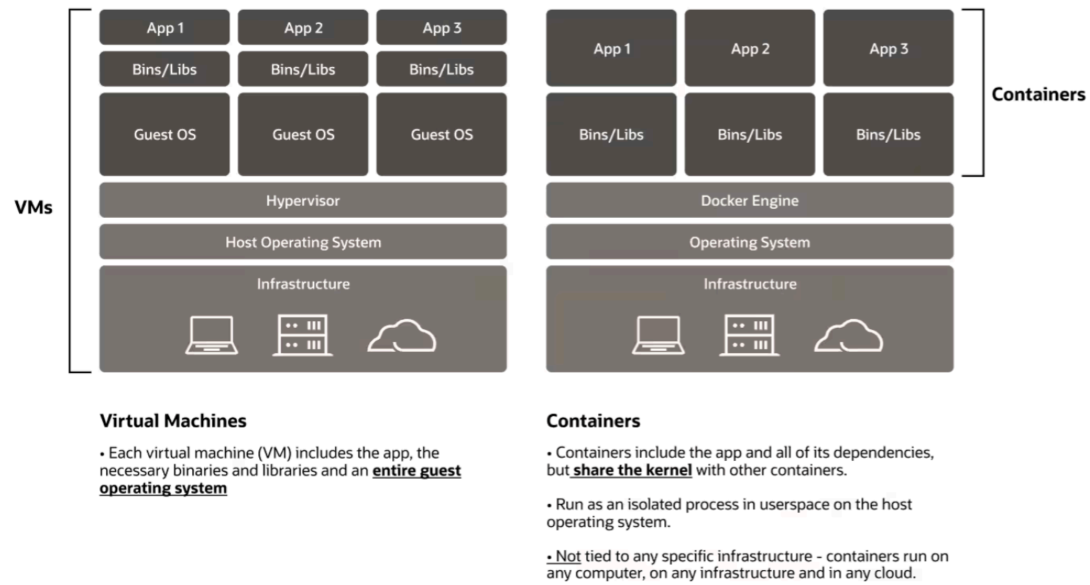
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	# OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

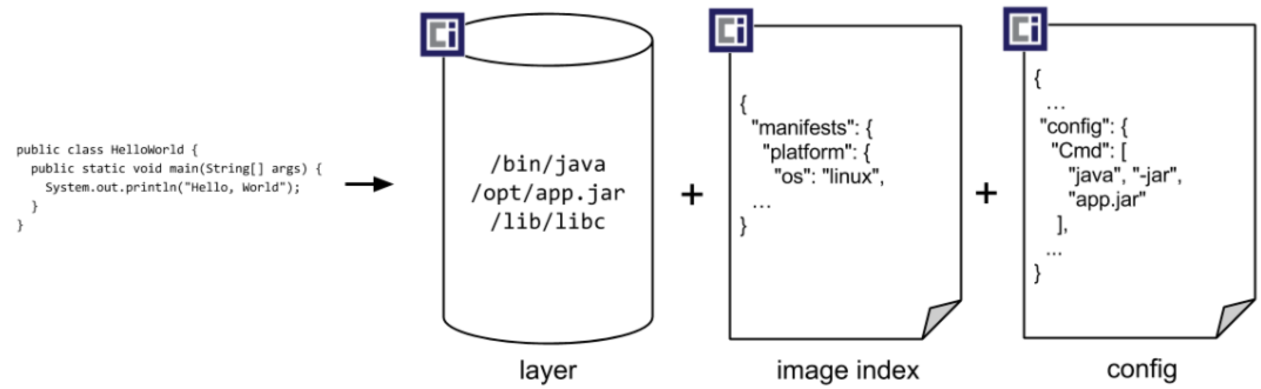
Claim 1	Accused Instrumentalities
	Layer • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md Containers only have access to resources that are defined in the image, https://www.hpe.com/us/en/what-is/docker.html

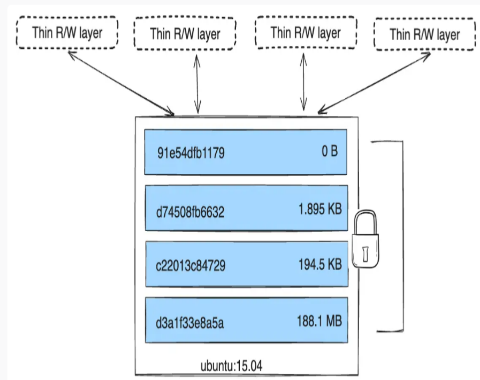
Claim 1	Accused Instrumentalities
	DESCRIPTION top `The programs ld.so and ld-linux.so* find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it.` https://man7.org/linux/man-pages/man8/ld.so.8.html
[1e] ii) wherein an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function, and	In each Accused Instrumentality, an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function. When a Docker or Kubernetes image is used to create a container in the Accused Instrumentalities, it creates a separate and isolated instance of a runtime environment which is independent of other containers running on the same host. Each container has its own instance of base images and its own data. The containers run in isolation, ensuring that the SLCSEs stored in the shared library are accessible to the software applications running in their respective containers. The image includes essential system files, libraries, and dependencies required to run the software application within the container. The containers can share common dependencies and components using layered images. This means that multiple containers utilize the same base image to create an instance. When an instance of SLCSE is provided from the base image (i.e., from the shared library) to an individual container including application software, it is not shared with other containers. This ensures that each container has its own isolated context. Docker or Kubernetes containers in the Accused Instrumentalities can share common dependencies and components using layered images. This means that multiple containers can utilize the same base image. Therefore, each container, containing the application software running under the operating system, utilizes a unique instance of the corresponding critical system element to execute the respective application software for performing a same function. *See, e.g.*:

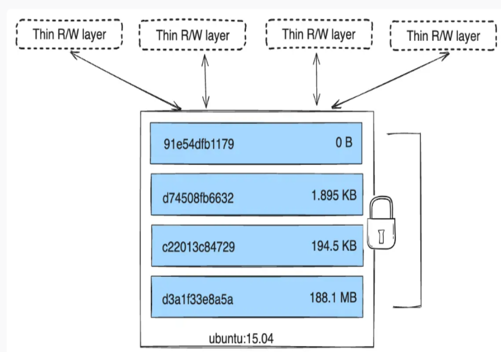
Claim 1	Accused Instrumentalities
	Cgroups and Namespaces History The underlying Linux kernel features that Docker uses are cgroups and namespaces. In 2008 cgroups were introduced to the Linux kernel based on work previously done by Google developers[1]. Cgroups limit and account for the resource usage of a set of operating system processes. The Linux kernel uses namespace to isolate the system resources of processes from each other. The first namespace, i.e. the mount namespace, was introduced as early as 2002.[2] https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/ **Container:** Unlike a VM which provides hardware virtualization, a container provides lightweight, operating-system-level virtualization by abstracting the "user space." Containers share the host system's kernel with other containers. A container, which runs on the host operating system, is a standard software unit that packages code and all its dependencies, so applications can run quickly and reliably from one environment to another. Containers are nonpersistent and are spun up from images. **Docker engine:** The open source host software building and running the containers. Docker Engines act as the client-server application supporting containers on various Windows servers and Linux operating systems, including Oracle Linux, CentOS, Debian, Fedora, RHEL, SUSE, and Ubuntu. **Docker images:** Collection of software to be run as a container that contains a set of instructions for creating a container that can run on the Docker platform. Images are immutable, and changes to an image require to build a new image. **Docker Registry:** Place to store and download images. The registry is a stateless and scalable server-side application that stores and distributes Docker images. https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/

Claim 1	Accused Instrumentalities
	Docker Basics The core concepts of Docker are images and containers. A Docker image contains everything that is needed to run your software: the code, a runtime (for example, Java Virtual Machine (JVM), drivers, tools, scripts, libraries, deployments, and more. A Docker container is a running instance of a Docker image. However, unlike in traditional virtualization with a type 1 or type 2 hypervisor, a Docker container runs on the kernel of the host operating system. Within a Docker image there is no separate operating system, as illustrated in Figure 1.  https://www.oracle.com/in/cloud/cloud-native/container-registry/what-is-docker/

Claim 1	Accused Instrumentalities
	# Setting Up Storage for Kubernetes Clusters *Find out how to define and apply persistent volume claims to clusters you've created using Kubernetes Engine (OKE). With Oracle Cloud Infrastructure as the underlying IaaS provider, you can provision persistent volume claims by attaching volumes from the Block Volume service or by mounting file systems from the File Storage service.* Container storage via a container's root file system is ephemeral, and can disappear upon container deletion and creation. To provide a durable location to prevent data from being lost, you can create and use persistent volumes to store data outside of containers. A persistent volume offers persistent storage that enables your data to remain intact, regardless of whether the containers to which the storage is connected are terminated. A persistent volume claim (PVC) is a request for storage, which is met by binding the PVC to a persistent volume (PV). A PVC provides an abstraction layer to the underlying storage. With Oracle Cloud Infrastructure, you can provision persistent volume claims: • By attaching volumes from the Oracle Cloud Infrastructure Block Volume service. The volumes are connected to clusters created by Kubernetes Engine using CSI (Container Storage Interface) or FlexVolume volume plugins deployed on the clusters. Oracle recommends the CSI volume plugin since the upstream Kubernetes project deprecates the FlexVolume volume plugin in Kubernetes version 1.23. See Provisioning PVCs on the Block Volume Service. • By mounting file systems in the Oracle Cloud Infrastructure File Storage service. The File Storage service file systems are mounted inside containers running on clusters created by Kubernetes Engine using a CSI (Container Storage Interface) volume plugin deployed on the clusters. See Provisioning PVCs on the File Storage Service. https://docs.oracle.com/en-us/iaas/Content/ContEng/Tasks/contengcreatingpersistentvolumeclaim.htm

Claim 1	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/ Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image
[1f] iii) wherein a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously.	In each Accused Instrumentality, a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously. For example, in Docker or Kubernetes containers used within the Accused Instrumentalities, each container operates independently, and a base image includes essential system files, libraries, and dependencies (i.e., SLCSEs) required to run the software application within the container. Based on information and belief, each element, such as system files, libraries, and dependencies (i.e., SLCSE) is associated with an execution of a predetermined function related to the application. When an image is used to create a container in the Accused Instrumentality, an instance of the SLCSE is provided to a

Claim 1	Accused Instrumentalities
	software application. Therefore, different instances of the SLCSE are provided to different applications for performing a same function, simultaneously. *See, e.g.*: Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state. https://docs.docker.com/get-started/overview/ Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/